RL-TR-96-137
Final Technical Report
July 1996

INCORPORATING A LEARNING CAPABILITY INTO THE KBSA

CTA Incorporated

Sidney Bailin

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

Rome Laboratory
Air Force Materiel Command
Rome, New York

This report has been reviewed by the Rome Laboratory Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS, it will be releasable to the general public, including foreign nations.

RL-TR-96-137 has been reviewed and is approved for publication.

APPROVED: Joseph A. Carozzoni

JOSEPH CAROZZONI
Project Engineer

FOR THE COMMANDER: John A. Graniero

JOHN A. GRANIERO
Chief Scientist
Command, Control & Communications Directorate

If your address has changed or if you wish to be removed from the Rome Laboratory mailing list, or if the addressee is no longer employed by your organization, please notify Rome Laboratory/ (C3CA), Rome NY 13441. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	July 1996	Final Aug 93 - Feb 96

4. TITLE AND SUBTITLE

INCORPORATING A LEARNING CAPABILITY INTO THE KBSA

5. FUNDING NUMBERS

C - F30602-93-C-0191
PE - 62702F
PR - 5581
TA - 27
WU - 71

6. AUTHOR(S)

Sidney Bailin

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

CTA Incorporated
Suite 800
6116 Executive Boulevard
Rockville MD 20852

8. PERFORMING ORGANIZATION REPORT NUMBER

N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Rome Laboratory /C3CA
525 Brooks Rd
Rome NY 13441-4505

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

RL-TR-96-137

11. SUPPLEMENTARY NOTES

Rome Laboratory Project Engineer: Joseph A. Carozzoni/C3CA/(315)330-7796

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

This report describes the research to investigate and demonstrate techniques for incorporating machine learning into the Knowledge-Based Software Assistant (KBSA) program. Advances in failure-based machine learning have made possible the incorporation of these techniques into advanced development paradigms, especially those dependent on acquisition of knowledge as part of the paradigm. The KBSA program, initiated in 1983, is a long-term program to introduce dramatic changes in the very nature of software development. The changes will arise through a new paradigm of development, based on a requirements-and-design-focused development activity. KBSA enables a shift in the level of development from the lower levels to the highest levels of life cycle through the formalisms used to specify those levels. Changes to the system are effected through transformations of the requirements or design specifications and implemented through automated mechanisms. This paradigm will dramatically improve both the quality and productivity of the software development process. One of the basic tenets of the program is to automate the assimilation of "knowledge" into the software assistant. The developer must be able to take advantage of prior events as stored with the KBSA. Knowledge acquisition and the automation of the process of acquiring knowledge are basic to this concept. Machine learning embodies the concept of acquiring knowledge that can be stored and effectively used in the KBSA.

14. SUBJECT TERMS

Parallel programming, Automatic programming, Formal methods, Knowledge-based systems

15. NUMBER OF PAGES

40

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

About Hendrix...[1]

Hendrix stands for Help Evaluating New Designs with Rules Interactively Extendible.

Hendrix was developed by CTA INCORPORATED under the Knowledge-Based Software Assistant program, sponsored by Rome Laboratory.

An earlier version of **Hendrix** was sponsored by NASA/Goddard Space Flight Center's Data Systems Technology Division. Hendrix makes extensive use of NASA's C-Language Integrated Production System (CLIPS).

The authors of **Hendrix** are:

Sidney Bailin and Dave Barker-Plummer	Rule base
Dick Bentz	User interface
Scott Henderson	Glue between user interface and rule base

Rome Laboratory Point of Contact:

Joseph A. Carozzoni
Rome Laboratory (USAF/AFMC)
Software Technology Division
525 Brooks Road, STE 57
Griffiss Air Force Base
NY 13441-4505
Email: jac@ai.rl.af.mil

Developers Point of Contact:

Sidney Bailin
Voice: (202) 328 - 3352
FAX: (202)328 - 3354
Email: sbailin@cta.com

Companion Volumes:

Hendrix Users Guide
Hendrix Implementors Guide
Hendrix Demonstration Guide

[1] This is Revision 0 of the Final Report for Hendrix Version 2.0..

Table of Contents

Table of Contents (continued)

List of Figures

1 Introduction

This document summarizes the work performed under a 30-month contract to investigate the feasibility of a learning capability within a Knowledge-Based Software Assistant. We begin, in this Introduction, by reiterating the goals of the project, describing the approach we took in carrying out the investigation, and stating the accomplishments of the project.

In Section 2 we present an overview of Hendrix, the tool developed as an experimental prototype and delivered as the practical outcome of the project. We discuss the motivation for the Failure-Based Learning approach, which was the focus of our research, and describe how it is implemented in Hendrix. We conclude this section with some examples of paraphrases of rules learned by Hendrix. The paraphrases are themselves generated by Hendrix; we include them here, rather than the formal rules themselves, because they are (by design) more comprehensible. This section is not meant to be a complete description or explanation of the Hendrix capabilities, but rather to provide a sense of what we have accomplished in the project. An accompanying document, the Hendrix Demonstration Guide, provides a much more detailed walk-through and explanation of the capabilities of the tool.

In Section 3 we address the specific accomplishments of the last phase of the project, in which we tried to address some of the problems with the preceding version of Hendrix, which was delivered in February, 1995. These problems, along with recommendations for addressing them, were identified in the Lessons Learned document dated April, 1995. We succeeded in implementing most, but not all, of the recommendations, and in this section we discuss where the tool still falls short, as well as our assessment of the effectiveness of the changes we implemented.

Finally, in Section 4, we discuss some directions in which this work could (and, we believe, should) proceed.

1.1 Project Goals

The underlying theme of this project has been the evolutionary nature of software development, and the need for an intelligent assistant to evolve along with the overall environment of end-users, developers, requirements, and software knowledge. Because the KBSA program has, historically, emphasized software development as a transformational activity — a series of transformations performed upon artifacts expressing or embodying desired properties of a software system, until an operational system is obtained — we have focused on the problem of evolving transformations. Each transformation represents knowledge of how to perform a certain software development task. The question, then, is whether some form of machine learning can allow an automated software assistant to acquire such knowledge on an ongoing, evolutionary basis, with minimal intrusion upon developers who are likewise continually learning and refining their knowledge.

Using the human expert-apprentice relationship as a model, we conjectured that one way to solve the problem would be to have the automated assistant learn by example. This would be minimally intrusive in that, when a developer has tried something new and discovered it works well, or has value, she could tell the automated assistant to generalize what has just been done and be ready to apply it to similar tasks in the future.

In our experience, much software knowledge is acquired in such an experimental fashion: the developer "discovers" an approach to solving a certain type of problem, the discovery often occurring in a more or less ad hoc fashion, driven by the requirements of the developer's current task, rather than as a theoretical development from known principles. An automated assistant that can learn from examples would be positioned to capture such knowledge, generalize it appropriately, and establish it as part of an evolving knowledge base, while otherwise it might never be codified, or might be codified for the use of the individual who discovered it but never be put into a form in which others could use it.

The goal of this project, then, has been to answer the following question: Can an automated assistant have the ability to learn significant (that is, useful and non-trivial) transformations from examples? In particular, can it have the ability to learn a design transformation from a single example of that transformation which would be provided by the developer, and to generalize it for future use?

Questions following from this primary question are: What specific capabilities would the automated assistant need in order to do this? What kind of input would it need from the developer teaching it the transformations, and how intrusive or difficult would it be for the developer to provide this input? And finally: Is there a chance that such an approach could provide real value in the real world? That is, could it make software development more manageable?

We should emphasize here that transformations are, in our opinion, not an "approach" to software development but rather a "view" of software development. Every software development task, from requirements analysis to maintenance, involves transformations of one sort or another. To direct our attention to automating transformations is not to advocate a particular approach to software development, but rather to appreciate the importance of transformations as a way of representing software knowledge.

1.2 Experimental Approach

We divided the period of performance into three phases of roughly equal length. The goal of each phase was to develop a learning capability along the lines described above, and evaluate its effectiveness; in each successive phase we attempted to overcome any problems discovered or lessons learned in the preceding phase.

We began with an assessment of the Concept Demo, which at the time this project began was the most developed and integrated form of KBSA technology. The Advanced Development Model (ADM), which is the successor to the Concept Demo, was just in its initial statges then, and it was not feasible for us to use it as the basis from which our work would proceed.

The Concept Demo gave us a sense of the types of specifications and transformations that the KBSA program had focused on, and the user interaction paradigms that might be expected of such a tool. One key trend, which we understood was to be carried further by the ADM, was a move towards specifications languages more typical of what is used widely in industry, and away from logic-based notations. Given this trend, and the goal of non-intrusiveness that we set ourselves, we decided (with Rome's concurrence) to use the Fusion object-oriented notation as the underlying specification language (Coleman *et al*, 1994) and to build Hendrix on a configurable graphical editor which had already been developed by CTA and had been used in a few other software tools. At the same time, we

took pains to create a well-defined, loosely coupled interface between the graphical editor and the underlying learning capability.

We set about looking for challenge problems that the learning capability would have to tackle. The very first version of Hendrix, which had earlier been developed for NASA/Goddard Space Flight Center, had been demonstrated only on "toy" problems. In this project we insisted on testing the newly developed capabilities on "real world" problems. We posted a notice in several Usenet news groups asking for suggestions of transformations that Hendrix might be challenged to learn. The most interesting response came from Eduardo Casais, who had recently received his Ph.D. in the area of transforming object-oriented inheritance graphs (Casais 1991). Casais had developed algorithms for improving the integrity and robustness of inheritance graphs, and he sent us literature describing this work.

We also consulted the literature on program transformations (e.g., Partsch 1990) and the work of Lieberherr's group on the Demeter methodology (Lieberherr *et al* 1988, 1991). Transformations described in these sources relied, to various extents, on program-level notations — for example, in the Demeter work, on explicit references to variables and function-call arguments. All of this work provides worthwhile challenges to the Hendrix approach, which we would like to pursue; but given the newness of our approach, the historical high-level orientation of the KBSA program, and our decision to use the Fusion notation as the specification language, we decided to avoid the finer-grained language level required by this other work. The Casais algorithms became, therefore, the central challenge problems with which we would test the Hendrix learning capability.[2]

The Casais algorithms are complex, and we started by asking why the original version of Hendrix would not be able to learn them. This led us to formulate certain generic capabilities that the learning capability would need — for example, the ability of the user to apply (i.e., compose) previously defined transformations when giving Hendrix an example of a new transformation.

Having formulated these requirements, we developed an initial capability which succeeded in learning a rule that, on paper at least, resembled the core Casais transformation. The first phase ended before we had a chance to run the learned transformation on examples in order to verify its correctness. The second phase of the project consisted mainly of discovering many mistakes in our approach (and in the resulting learned transformations) and refining the learning algorithm to produce better rules.

We succeeded, by the end of the second phase, in teaching Hendrix the core transformation and verifying its behavior on some examples. In the process, however, we had introduced a number of cludges which were not consistent with our goals of usability and non-intrusiveness. Some logical limitations also became apparent. All of these issues were written up in the Lessons Learned document. The third and final phase of the project has been devoted to implementing and testing the recommendations made in that document.

1.3 Accomplishments

We have implemented most, but not all, of the recommendations in the Lessons Learned document. Those that we did not implement do not pose any unusual development challenges — we simply ran out of time. The changes that we succeeded in implementing

[2] Any indication of "hard wiring" the capability to support these algorithms would, of course, invalidate our work. The documentation shows, we believe, that we have not built any such bias into the tool.

have allowed us to simplify the Hendrix Demonstration considerably; for example, roughly half the number of diagrams (28 as opposed to 54) are now required to teach Hendrix the same transformation.

On the basis of the current Hendrix capabilities, the answer to the questions posed under Project Goals is, we submit, a qualified Yes. We have shown how an automated assistant can learn a complex, real-world design transformation by abstracting from a single example. We believe this is evidence that the capabilities postulated in those questions are achievable. But the system is still a prototype, and further experimental work would be needed to create a truly usable learning-based automated assistant:

- It is still difficult identifying all of the basic transformations that have to be taught to Hendrix as components of a complex transformation. Some of these require envisioning intermediate states of the complex transformation which are not immediately obvious.

  This problem has a solution, which we would have implemented had we the time. It involves a form of number generalization which would allow the user to subsume, under a single transformation, steps that currently have to be defined individually. This is discussed in Section 4.1.

- The system needs to be challenged with a much broader set of transformations to be learned. It was our intention to do this in the final phase of the project, but again we ran out of time.

- When the level of granularity of the specification language is lowered, to enable the formulation of program-level transformations such as those discussed in Section 1.2, there will probably be a new challenge for Hendrix: deciding what information in an example is relevant to the intended transformation, and what may be abstracted out. In the field of explanation-based learning, this is known as irrelevant feature elimination. It is likely to be an important capability if Hendrix-like techniques are to be used to support program maintenance at the source code level.

These issues are discussed further in Section 4. Also in that section we discuss the possibility of using Hendrix to learn how to generate code. We performed some initial experiments in this area and the results are promising. These experiments indicated that the number generalization capability, mentioned above, is likely to be necessary for a user-friendly tool that can learn code generation rules from single examples.

Finally, a paper describing in detail the capabilities and implementation of Hendrix (in its Phase Two form) was submitted to the Journal of Knowledge Based Software Engineering, and is currently being refereed.

2 Overview of Hendrix

Hendrix is an experimental tool for exploring the role of learning in automated software development. Learning is a key part of the evolution of software development knowledge; Hendrix shows how an automated assistant can learn along with its user, as approaches to software development change.

Hendrix is a transformational tool. It assists the software developer by recognizing *patterns* in requirements or design specifications. Where appropriate, Hendrix *transforms* these into other patterns. Reasons for transforming a pattern might be:

- The original pattern fails some quality or correctness criterion
- The transformed pattern is closer to implementation

Hendrix learns patterns and transformations by example. The user shows it an example of a pattern, and Hendrix generalizes it into a rule for recognizing the pattern in other specifications. The user transforms an example of the pattern, and Hendrix generalizes this into a rule for transforming the pattern in other specifications.

2.1 Motivation for a Learning Capability

We tend to use software to perform novel, unprecedented functions. Even in a mature application domain in which a substantial body of technology is reused, new kinds of requirements may still be levied on the next system to be built. And it is not only requirements that evolve — solution techniques evolve too. A software assistant, therefore, must not be locked into a fixed theory of how software is developed. In a transformation-based KBSE, learning can help in performing the following ongoing tasks:

- Defining an adequate set of transformations
- Knowing under what conditions to apply a given transformation
- Knowing the criteria for accepting the results of a transformation
- Classifying transformations in terms of their purpose and applicability

2.1.1 Motivation for Failure-Based Learning

Henry Petroski, in his book *To Engineer is Human,* argues that engineering failures are a key impetus for learning (Petroski 1992). His book analyzes in detail several well-known engineering disasters, and demonstrates how the respective engineering fields have advanced as a result of the lessons learned. In the field of software engineering, an analogous source of information is the "Risks to the Public" column in the ACM *Software Engineering Notes.*
Obviously a software assistant's learning capability should not be predicated on the occurrence of disasters, but rather on the occurrence of *errors* which are, like it or not, a fundamental part of the development process. The notion of learning from experience is key to the process improvement paradigm articulated by Basili in his Experience Factory proposal (Basili 1993).

The paradigm of Failure-Based Learning (FBL) is that when an error occurs, and is recognized, a transformation from the erroneous situation to a corrected situation should be

learned. Learning is accomplished by generalizing the "fix" that has been applied to the current situation.

2.2 The Hendrix Tool

Hendrix is a prototype implementation of FBL in an interactive software specification and design assistant. Specifications and designs are created through an graphical editor that has been configured to support the notations of the Fusion methodology. Intelligent assistance is provided by the Hendrix capability to evaluate and transform specifications. FBL is implemented in the capability of Hendrix to learn how to recognize and transform patterns in a specification.

Hendrix is written in C with the CLIPS run-time library linked in. It was developed under SunOS 4.3 and uses the Motif Window Manager. Although the tool will run with other window managers, it relies on a window-close operation being available to the user through the window manager. This is an unfortunate limitation which would, however, be trivial to fix.

From the user's point of view, Hendrix consists of three main parts: the persistent store, the intelligent editor, and the software assistant.

2.2.1 The Persistent Store: Specifications and Rules

The persistent store contains both application data — the specifications of software that the user and Hendrix cooperate to develop — and meta-knowledge, or rules concerning the development of such specifications.

Specifications in Hendrix take the form of Fusion object-oriented diagrams. Five types of Fusion diagrams are supported: Object Model, Inheritance Graph, Visibility Graph, Object Interaction Graph, and Timing Diagram. In addition, a simple State Transition Diagram is included although it is not part of the Fusion method per se. Hendrix stores these models together as different views of the same specification, with like-named nodes treated as representing the same entity.

We adopted Fusion because it is a synthesis of many of the best aspects of other object-oriented approaches, and at the same time it is relatively simple. One can, however, use Hendrix to define concepts and rules over and above the Fusion notation, and even to reinterpret the notation.

Meta-knowledge in Hendrix takes the form of production rules in the CLIPS language. There are three types of rules: *concept rules* define predicates for talking about specifications, *evaluation rules* define criteria against which specifications can be tested for conformance, and *transformation rules* define transformations that operate on (and yield) specifications. Concept rules and evaluation rules are both instances of *pattern rules*, i.e., rules for recognizing various patterns in a specification. Concept rules and evaluation rules are learned by the same internal mechanism in Hendrix. To the user, however, they play different roles.

2.2.2 The Intelligent Editor

The user creates a specification by drawing and annotating diagrams. The diagram editor is always in front of the user while Hendrix is running. Every editing event — adding, moving, labeling, deleting, and annotating nodes or arcs — is passed on to the rule base, which maintains a current view of the state of the specification and can therefore react as needed.

Figures 2-1 through 2-3 illustrate the relationship between the diagram editor and the rule base. In Figure 2-1, the user has created an Object Model (`om`), drawn two nodes, and connected them with an arc. Figure 2-2 shows how nodes are labeled and annotated with attributes. Each such event is passed to the rule base as a `user-action` fact, which then gets integrated into the facts representing the current diagram state (Figure 2-3).

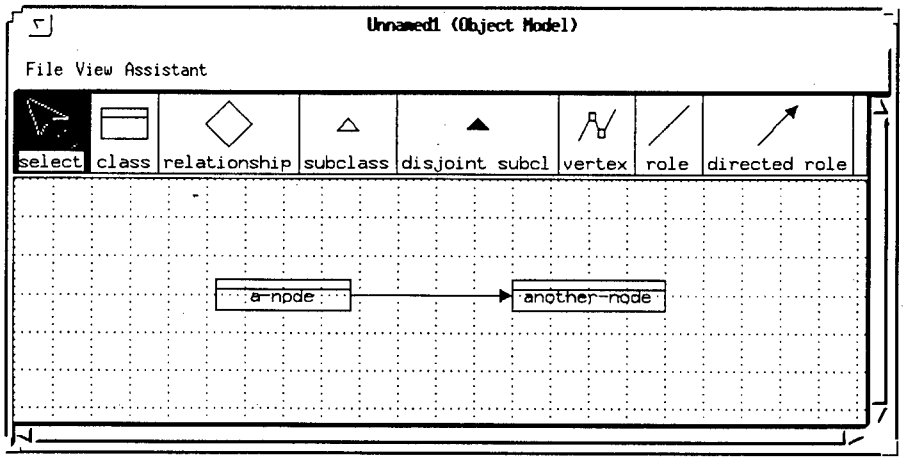

Figure 2-1: The user interacts with the diagram editor at all times.

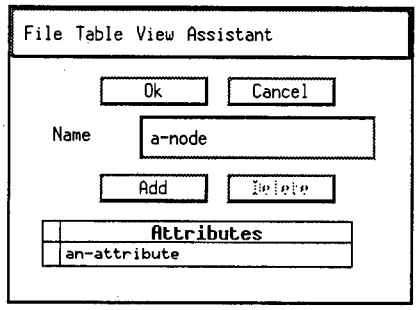

Figure 2-2: Nodes are labeled and annotated in a pop-up window.

```
FIRE    1 add-node: f-97
<== f-97    (user-action om AddNode class 1 at 176 132 size 32 91)
==> f-98    (assert-this om class no-label1 1 "176,132" "32,91")

FIRE    1 label-node: f-101,f-99
<== f-101   (user-action om LabelNode class 1 a-node)
==> f-102   (retract-this om class no-label1 1 "176,132" "32,91")
==> f-103   (assert-this om class a-node 1 "176,132" "32,91")

FIRE    1 add-attribute-to-node: f-105,f-104
<== f-105   (user-action om addAttribute class 1 an-attribute)
==> f-106   (assert-this om attribute a-node 1 gen2 . an-attribute)

FIRE    1 add-node: f-108
<== f-108   (user-action om AddNode class 2 at 376 132 size 32 105)
==> f-109   (assert-this om class no-label3 2 "376,132" "32,105")

FIRE    1 label-node: f-112,f-110
<== f-112   (user-action om LabelNode class 2 another-node)
==> f-113   (retract-this om class no-label3 2 "376,132" "32,105")
==> f-114   (assert-this om class another-node 2 "376,132" "32,105")

FIRE    1 add-arc: f-116,f-104,f-115
<== f-116   (user-action om AddArc directed-arc 4 from 1 to 2)
==> f-117   (assert-this om directed-arc no-label4 4 1 2)
```

Figure 2-3: Editing events are passed to the rule base as facts.

2.2.3 The Software Assistant

The assistant is the heart of the tool and the locus of its innovative aspects. Hendrix performs the following software assistant functions:

- *Evaluate* a specification
- *Transform* a specification
- *Learn* new concepts, evaluation rules, and transformations

Evaluation is the means by which Hendrix acts as a critic and advisor to the user. Upon evaluating a specification, Hendrix may report back constraint violations or simply recommendations for further action. Evaluation is a pattern recognition process; the patterns to be recognized are taught to Hendrix by example.

Transformations are the means by which Hendrix acts as an assistant. The purpose of a transformation may be to fix a problem, or simply to carry the software development forward by moving the specification closer to implementation. The user initiates all transformations, choosing from a menu of transformations that Hendrix is capable of performing. This menu grows as Hendrix is taught new transformations.

Transformations may be correlated many-to-one with evaluation rules. For each transformation, there is an initial pattern and a resulting pattern. The initial pattern is identified through the evaluation rule that recognizes the pattern. The same pattern may, however, be transformable in several different ways.

Concepts are predicates that say something *about* a specification. Like evaluation rules, they too are represented as patterns to be recognized. Unlike evaluation rules, however, concepts extend the *language* for talking about specifications — i.e., the language in which patterns themselves are expressed. This capability enables the user to evolve a design theory over time, building layers of increasingly rich abstractions on the basis of experience and lessons learned.

Learning in Hendrix involves presenting an example of a pattern, or of the transformation of a pattern, and having Hendrix generalize it into rules that apply to other cases of the pattern. This is quite different from *empirical learning* in the sense that the machine learning community uses that term, but it bears a close resemblance to *explanation-based learning* (DeJong and Mooney 1986).

2.3 Implementation of Failure-Based Learning in Hendrix

The Hendrix Demonstration Guide presents a complete and detailed example of teaching Hendrix a complex design transformation. In this section we summarize the algorithms Hendrix uses to generate pattern and transformation rules from examplar specifications.

2.3.1 Learning Evaluation Rules

Learning an evaluation rule involves the following key steps:

1. Take the *base facts* of the example
2. *Variablize* the elements in these base facts
3. Generate a *binding list* to be asserted by the rule

Base facts. The base facts in a specification are those that are explicitly represented — as opposed to those that Hendrix can infer. For example, in Figure 2-4 there are two base facts, `(subclass A B)` and `(derived-class B C)`, which would be included as conditions in the generated rule. From these facts, Hendrix can infer the additional fact `(derived-class A C)`. The inferred fact, however, would not be included in the conditions of the generated rule, since it is not explicitly represented in the example.

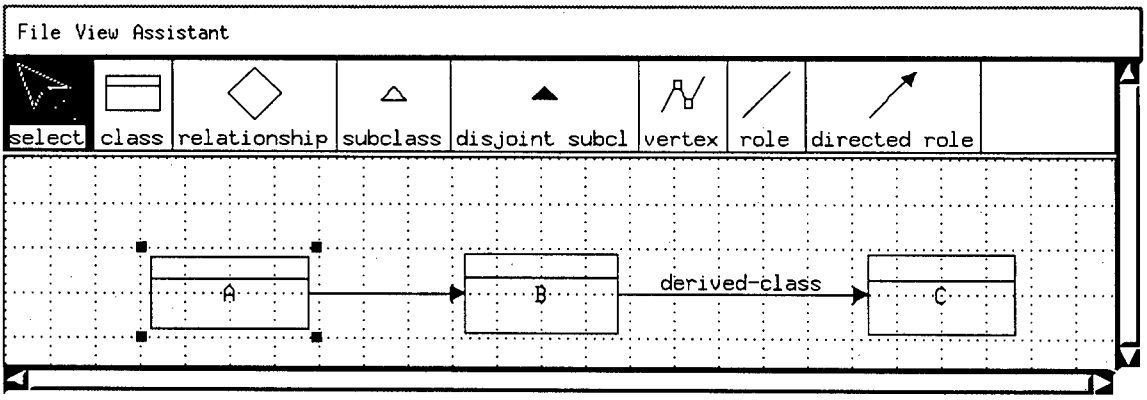

Figure 2-4: Base facts are those that are explicitly stated in the diagram.

Variablization. Variablization replaces the names of particular elements in the current specification with variables. The resulting variables can then be bound to "corresponding" elements in other specifications.[3] A name can consist of multiple "tokens" (e.g., "cat dog mouse") and each token can be constrained to satsify a regular expression (see Section 3.1). A distinct variable is generated to represent each token.

Each variable is automatically constrained to take a value distinct from all other variables, unless the user specifies otherwise (see Section 3.2). If a token appears more than once, the same variable takes its place in each occurrence. Distinct names can, therefore, share components (e.g., "Felix the Cat" and "Socks the Cat").

Binding list. Patterns and transformations are learned as separate rules. The separation enables the Hendrix user to detect patterns — e.g., constraint violations — without necessarily performing a transformation. It also enables the user to teach Hendrix distinct transformations of a single pattern.

Despite their separation, the two types of rules are related: transformations act on previously detected patterns. There must, therefore, be a way for an evaluation rule to communicate information about a pattern to a subsequently executed transformation rule. The binding list is the medium of this communication.

The binding list is a set of pairs of the form:

```
role-name  instance-identifier
```

There is a pair in the list for each element of the pattern. The list is asserted as a fact by the evaluation rule, and occurs as a pre-condition in the transformation rule, thus enabling communication between the two rules. In both the evaluation and transformation rules, `role-name` is a literal, while the `instance-identifier` is a variable which takes on the appropriate value in the current specification when the rule fires.

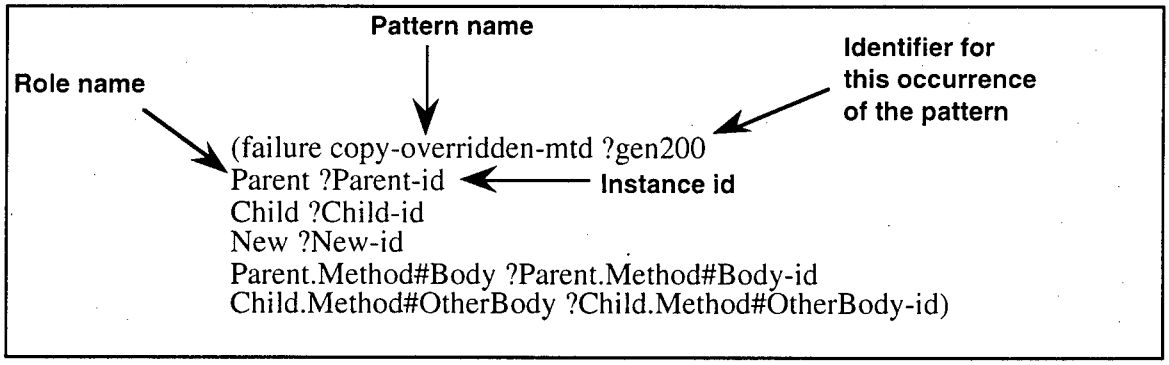

Figure 2-5: The binding list serves as a bridge between evaluation and transformation rules.

[3] Variablization is analogous to *identity generalization* in Explanation-Based Learning.

2.3.2 Learning Transformations

Transformation rules are learned by comparing the resulting pattern with the original pattern in an example that the user has provided. The differences between the two patterns are expressed as a set of facts:

- Base facts that are true in the initial pattern but not in the transformed pattern
- Base facts that are true in the transformed version but not in the initial pattern

Hendrix infers from these facts a set of `assert` and `retract` operations that, taken together, effect the transformation. The elements in the `assert` and `retract` operations are then variablized to apply to all future instances of the pattern.

Variablizing a transformation. Variablizing the `asserts` and `retracts` that constitute a transformation relies in an essential way on the binding that was asserted by the evaluation rule. The process consists of two steps:

1. Given an element involved in the transformation, locate it in the binding list, and retrieve the corresponding role name
2. Convert the role name into the corresponding variable name used in the evaluation rule

Thus, the role name serves as a pivot in converting an *instantiated variable* into the *name of the variable* itself, as shown in Figure 6:

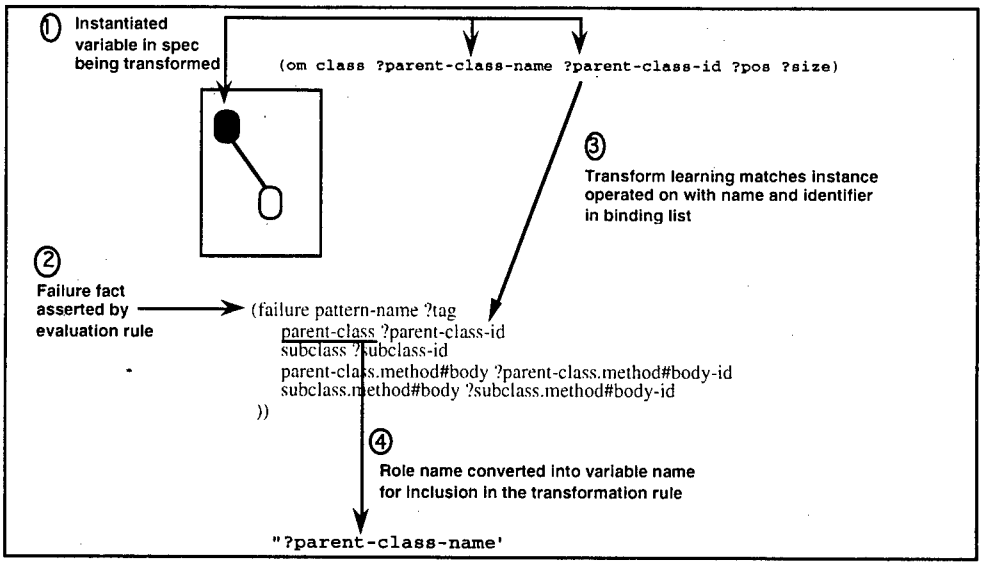

Figure 2-6: The binding list is used to infer variable names from their current values

There are three complicating factors in variablizing a transformation:

Ambiguous elements. The user might have modified an element that plays more than one role in the pattern. Hendrix has no way of inferring, in the general case, which of these roles to be the object of this particular modification. The situation is probably an indication that the example is not an appropriate one for teaching this transformation. An

appropriate response might be to report the problem to the user, and either abort the learning process or allow the user to clarify his intention. In the current implementation, however, Hendrix simply chooses the first role that it finds played by the affected element. The automatically generated English paraphrase of the resulting rule will make clear to the user whether his intentions were satisfied (see Section 3.2.1).

Anomalous elements. The user might have modified an element that did not play a role in the inital pattern. In this case, there is no way to generalize the transformation, so Hendrix informs the user and aborts the learning process.

New elements. Complexities arise when Hendrix learns a transformation involving the *creation* of a node, arc, or attribute. The complexities occur because transformations can be iterated as well as composed; thorugh either of these operations, the transformation may be applied to multiple instances of a pattern in a single (possibly evolving) specification.

Applying the transformation to multiple instances raises the following question: should a new element be created for *each* instance of the pattern, or should only *one* new element be created for *all* instances? Or, more generally, are the instances logically grouped in such a way that each *group* requires a new element to be created?

Hendrix handles this situation by assuming that the new element is a function of some — possibly all — of the elements in the initial pattern, and asking the user to specify this subset. The user is presented with a matrix whose rows correspond to the new elements, and whose columns correspond to the original elements in the pattern. Hendrix asks the user to define each new element as a function of zero, one or more of the original elements.

In adopting this approach, we rejected an alternative approach which might at first seem more intelligent, but which on closer analysis proves only to be more error prone. The deceptively intelligent approach is to have Hendrix infer the appropriate functionality from multiple examples of the transformation.[4]

Such a process would require the user to present Hendrix with a consistent and sufficient set of examples. Hendrix could easily recognize inconsistency or incompleteness in the example set, but would be unable proceed in either of these circumstances. The burden of providing an appropriate set of examples would therefore remain with the user, who in doing so would have to bear in mind the very information that Hendrix now prompts for.

We concluded from these considerations that — although it appears to be more in spirit with the learning process — there is really no advantage to having Hendrix infer the functional dependencies.

2.4 Some Paraphrased Examples of Learned Rules

The Hendrix pattern language, which consists of the Fusion diagrams plus a small number of meta-constructs, allows the user to specify quite complex patterns by example. The meta-constructs allow the user to:

- Refer to other patterns in a pattern being defined
- Define inductive patterns by recursion (self-reference)
- Negate and universally quantify over elements in a referenced pattern

[4] This would be analogous to *number generalization* in Explanation-Based.

The following figures show some examples of pattern rules learned by Hendrix. The English paraphrase, which we show here, is generated at the same time that the CLIPS rule is generated. In the Demonstration Guide we discuss these and other examples in more detail, presenting both the diagrams from which they were generated and the resulting CLIPS rules.

The first example is the step (recursive) case of the concept "derived," where one class is derived from another if there is a chain of subclass links between the two. We have used an unlabeled directed arc to signify the Parent-Subclass relation (the base case of the concept), and this rule extends the definition to arbitrarily long chains. In other words, if Parent is derived from Ancestor, and Child is an immediate subclass of Parent, then Child is also derived from Ancestor:

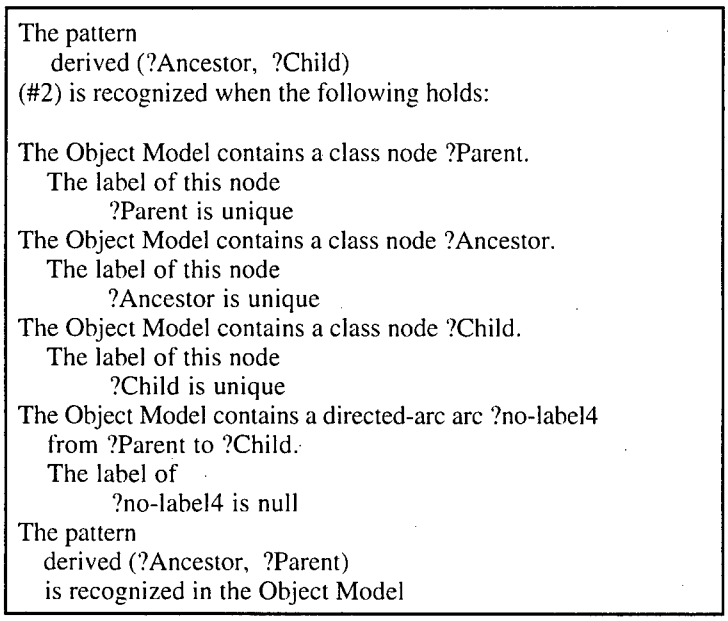

Figure 2-7: A pattern may reference itself to achieve recursion.

The next example, which specifies when two classes have different ancestors, illustrates the use of negation:

```
The pattern
   different-ancestors (?Non-Descendant, ?Descendant)
(#1) is recognized when the following holds:

The Object Model contains a class node ?Ancestor.
   The label of this node
         ?Ancestor is unique
The Object Model contains a class node ?Descendant.
   The label of this node
         ?Descendant is unique
The Object Model contains a class node ?Non-Descendant.
   The label of this node
         ?Non-Descendant may be the same as ?Ancestor
The pattern
   derived (?Ancestor, ?Descendant)
   is recognized in the Object Model
The pattern
   derived (?Ancestor, ?Non-Descendant)
   is not recognized in the Object Model
```

Figure 2-8: A pattern may reference the negation of another pattern.

As a last example of a learned pattern, we show the paraphrase of a rule that recognizes when a class introduces a method, i.e., no ancestor class possesses the same method. This pattern illustrates the use of negation and universal quantification:

```
The pattern
   introduces-method (?Intro-Class)
(#1) is recognized when the following holds:

The Object Model contains a class node ?Intro-Class.
   The label of this node
         ?Intro-Class is unique
   Node ?Intro-Class has the following attribute(s):
      ?Method ?Body
      where:
         ?Method is unique
         ?Body is unique

For every class node ?Class-With-Same-Method in the Object Model
   whose label
         ?Class-With-Same-Method is unique
   and which has the following attribute(s):
      ?Method ?Body
The pattern
   derived (?Class-With-Same-Method, ?Intro-Class)
   is not recognized in the Object Model
```

Figure 2-9: A pattern may universally quantify the variables of another pattern.

Finally we give the paraphrase of the complex transformation which motivates all of the examples in the Demonstration Guide. This example illustrates most of the features of transformation learning in Hendrix: removal of elements from the original pattern, creation of new elements as a function of (a proper subset of) the original elements, composition of transformations, recursion, and focusing the attention of a composed transformation on a

subset of elements — some of which were in the original pattern and some of which were created by previous steps in the transformation

In the following figure, the single asterisk (*) points out an instance of defining new elements as functions of already existing elements. The double asterisk (**) indicates an example of focusing attention by restricting the bindings of an iterated transformation. The triple asterisk (***) indicates a recursive call of a transformation to itself.

```
The pattern Casais(Parent, Child, no-label9, Parent.m#b1,  Child.m#b2)
   may be transformed as follows:

 Remove the directed-arc no-label9 from the Object Model.
 If there isn't one already, create a class node
   ?New(?Child,  ?Parent)    in the Object Model                          *
   where:
         ?New is new
 If there isn't one already, create a directed-arc arc
   ?no-label79(?Child,  ?Parent) from ?New to ?Child in the Object Model
   where:
         ?no-label79 is null
 After these operations, perform the following transformations:
      repair-copy-overridden-mtd* with:
          Parent restricted to Parent
          Child restricted to Child
          New restricted to Casais#New[Child,Parent]                      * *
      repair-copy-disembodied-mtd* with:
          New restricted to Casais#New[Child,Parent]
          Child restricted to Child
          Parent restricted to Parent
      repair-copy-inherited-method* with:
          Parent restricted to Parent
          Child restricted to Child
          New restricted to Casais#New[Child,Parent]
      repair-copy-virtual-method* with:
          New restricted to Casais#New[Child,Parent]
          Parent restricted to Parent
      repair-copy-grandparent* with:
          Parent restricted to Parent
          New restricted to Casais#New[Child,Parent]
      repair-Casais-partial*                                              * * *
      repair-make-parent-a-subclass* with:
          Parent restricted to Parent
          New restricted to Casais#New[Child,Parent]
      repair-remove-grandparent-link* with:
          New restricted to Casais#New[Child,Parent]
          Parent restricted to Parent
```

Figure 2-10: Transformations may involve functional dependencies, recursion, and focused bindings

A detailed discussion of this and the component transformations, the patterns they transform, and the procedures by which Hendrix learns them, is provided in the Hendrix Demonstration Guide.

3 Progress Since the Lessons Learned Document

In this section we summarize the work performed in the last phase of the project, which has concentrated on implementing the recommendations of the Lessons Learned Document, written in April, 1995. The recommendations fell into three broad categories, dealing with the language for specifying pattern exemplars for Hendrix to generalize, the ability of the user to interact with the Hendrix pattern base, and the efficiency of the Hendrix transformation engine.

3.1 Extending the Pattern Language

In the Phase Two demonstration we were able to express some fairly complex concepts by example, but there was a lot of overhead in the process. The patterns had to be decomposed into numerous special cases, and logical constructs required a rather fine-grained layering of examples on top of each other by reference. The recommendations made in the Lessons Learned documents aimed to eliminate these difficulties.

3.1.1 Regular Expressions

Many of the limitations encountered in Hendrix 1.0 (the Phase Two version) concerned the ad hoc way in which names and attributes were handled in pattern exemplars. Names were basically treated as variables, with a notation to force them to be treated as constants (literals), and another notation to express the requirement that a name *not* be a specified literal. These conventions were not handled uniformly, however. Arc names were always treated as literals. Furthermore, the notation for negation had a different interpretation for arcs: on a plain arc, if the label of the arc was the name of a concept known to Hendrix, then negation meant that the concept should not hold.

The major recommendation was to support a general notation for regular expressions in labels, uniformly over nodes, arcs, and attributes. A Hendrix label can now be of the form `tag:regex`, where `tag` is the name of the pattern element and `regex` is a regular expression which the element's name must match in instances of the pattern. Nodes, arcs, and attributes are now treated uniformly; in particular, arcs and attributes are now treated as first-class objects and are not subject to any special restrictions.

We implemented regular expression matching by constructing an interface with perl, which provides this as a built-in capability. The interface in its current form is less than streamlined, involving a separate Unix `exec` for each call to perl. A more efficient implementation would be to start up a perl process when Hendrix starts up, with pipes between the two processes: the candidate labels and constraining regular expressions would be passed to perl over one pipe, and perl would similarly pass its output back over the other. Modifying the current implementation to do this is something that any Unix application hacker can easily do.

Another approach, which we seriously considered, was to link in the Gnu library of C functions supporting regular expressions. We rejected this approach not for technical reasons but because of the Gnu requirements for sharing software that incorporates Gnu code. We did not know whether this would conflict with Government rights to the data,

and did not feel there was enough time to explore this issue. Technically, it is probably the preferred approach.

3.1.2 Pegs

Another limitation of Hendrix 1.0 concerned the automatic assumption, in the generated rules, that distinct items in an exemplar pattern diagram necessarily represented distinct items in all instances of the pattern. This assumpmtion again forced us to develop several special cases of patterns which intuitively should have been subsumed under one pattern definition. Hendrix 2.0 addresses this problem through the use of "pegs."

Pegs are additional annotations to a label (or attribute token) in a pattern exemplar. The pegs identify other labels in the pattern that may take the same value, in instances of the pattern, as this element. For example, a pattern that contains two nodes, one labeled A and the other labeled `B?=A`, indicates that `B` may be the same as `A`. The pattern will therefore match specifications containing one or two nodes. Without the peg, i.e., in a pattern exemplar containing nodes labeled `A` and `B`, Hendrix will generate a pattern recognition rule that requires nodes `A` and `B` to be distinct.

This notation enabled us to eliminate diagrams in the Hendrix Demonstration that were needed to include special cases of a pattern in which a single node played two roles. For example, the concept `different-ancestors` is exemplified by a pattern that shows three classes: an ancestor, a descendant of that ancestor, and another class which is not a descendant of that ancestor. The third class may be the same as the ancestor, and in Hendrix 1.0 this case had to be specified in a separate diagram, and a separate rule learned from it. In the current version, a peg suffices to indicate that the ancestor and the non-descendant may or may not be the same class.

3.1.3 Universal Quantification

Hendrix 1.0 supported a specific case of universal quantification, but it was apparent that a general capability would simplify teaching the system certain concepts. In the Lessons Learned Document we recommended that a general capability to universally quantify over diagram objects be implemented. This has been implemented using the approach suggested in that document, essentially treating the universal quantification `forall A.P(A)` as `not there-exists A.not P(A)` and implementing this, in turn, using CLIPS negation. In CLIPS 6.0 there is explicit support for universal quantification, so Hendrix 2.1 takes advantage of this, which simplifies the generated rules.

The support for universal quantification was directly responsible for eliminating some of the pattern exemplars needed in the Phase Two demonstration. For example, in Hendrix 1.0 we had to define the concept `introduces-a-method` by negating the concept `inherits-method`. The diagram exemplifying `inherits-method` was not needed in the demonstration for any other purpose. In Hendrix 2.0 we can define `introduces-a-method` directly using universal quantification, as shown in the rule paraphrased in Section `2.4`.

3.1.4 Arbitrary Number of Tokens in Labels and Attributes

Hendrix 1.0 required node and arc labels to consist of a single word, or token, and attributes to consist of one or two tokens. Besides being unnaturally restrictive (especially for attributes), the non-uniformity resulted in a great deal of conditional logic in the Hendrix 1.0 implementation to distinguish attributes from nodes and arcs and to distinguish single-token attributes from those with two tokens. In the current version, node and arc labels as well as attributes may consist of an arbitrary number of tokens, and similarly a concept may have an arbitrary number of arguments.

Although this feature was not needed in our revision of the Hendrix Demonstration, we believe it will prove useful in teaching Hendrix other patterns and transformations. Labels can now be distinct and yet share one or more tokens between them. This enables the user to indicate relationships between design elements through their labels. For example, in the Lessons Learned document we noted the inability in Hendrix 1.0 to specify the following evaluation rule from the Shlaer-Mellor methodology:

> An associative object may have an additional state model to manage the creation of instances of the associative object. The name of this state model is <object>_ASSIGNER where <object> is the name of the associative object...[5]

In Hendrix 2.0, the user can specify a pattern in which one node has the label `<object>` and another has the label

```
<object> :ASSIGNER
```

(with a space rather than an underscore between them). The literal token ASSIGNER is specified as a constant regular expression, and the generated pattern rule will look for this literal as the second token of any node playing this role in an instance of the pattern.

3.2 Organizing Patterns

The Lessons Learned Document recommended several capabilities to help the user manage an ever-expanding set of interrelated pattern definitions. The primary motivation for these capabilities was the recurrent need, in the Phase Two demonstration, to define subtle variants of patterns; it was difficult to assess, at any given moment, whether all necessary cases of a concept had been covered by the patterns defined so far. We made the following recommendations:

- Integrate concept and evaluation rules into a single type of pattern rule
- Organize patterns into a type hierarchy
- Provide a paraphrase capability
- Provide information about partial matches of a specification to a pattern

[5] Rule number 16 in N. Lang, "Shaer-Mellor Object-Oriented Analysis Rules," *Software Engineering Notes*, Vol. 18 No. 1, January 1993.

3.2.1 Integration of Concept and Evaluation Rules

The two files, `hendrix.learn-concept` and `hendrix.learn-eval-rule`, have been replaced in Hendrix 2.0 by a single file, `hendrix.learn-pattern`, which is responsible for learning both concept and evaluation rules. The treatment of both types of rules is identical except for the respective conditions under which they fire and the form of the facts that they assert. Concept rules can now fire at any time, thanks to truth maintenance (see Section 3.3.2), while evaluation rules fire only upon user request or when a transformation is being performed. Concept rules assert facts that can be matched as conditions of other concept rules, while evaluation rules assert failure conditions that can be matched as conditions for transformation rules. A nice simplification of the entire system has resulted from the elimination of pseudo-evaluation rules, whose function is now performed by the evaluation rules themselves.

We intended, but did not have time, to fold `hendrix.cases` into this module as well, since in Hendrix 1.0 case-distinction rules were generated using the same approach as concept and evaluation rules. In the distribution of Hendrix 2.0, the file `hendrix.cases` exists but is not loaded, since it has not been updated to accommodate changed fact and rule formats. The case-distinction rule generation capability should be added to `hendrix.learn-pattern`.

3.2.2 Organizing Patterns into a Type Hierarchy

Time limitations prevented us from doing anything on this item. Preliminary analysis did make clear that the type hierarchy itself would typically be quite shallow. The most important capability would be to assist the user in visualizing the reference relationships among patterns, and in defining new patterns through logical operations on existing patterns.

3.2.3 Paraphrase Capability

Hendrix 2.0 generates English paraphrases of pattern and transformation rules at the same time that it generates the rules themselves. The paraphrase capability is sensitive to many of the expected uses ("cliches") of the extended pattern language, and we believe it generates quite understandable output. This has become even more important in Hendrix 2.0, as the generated rules themselves have become more complex to accommodate the extended pattern language and truth maintenance.

3.2.4 Reporting Partial Matches

In the Phase Two demonstration, we frequently found ourselves expecting a learned pattern to be recognized in a specification, and were surprised when it was not. We therefore recommended a capability that would report which conditions of a pattern were satisfied by a specification, and which were not.

Such a capability is analogous to the built-in CLIPS `matches` function. It is different, however, because the individual CLIPS conditions in a Hendrix pattern rule are at a lower level than the conceptual conditions that the pattern rule is intended to express (and that are summarized by the paraphrases). For example, in a pattern that contains two classes A and

B where B is stipulated to be derived from A, the "derived" relationship between A and B is expressed in the following CLIPS conditions:

```
(known-concept ?concept-name)
(?concept-name ?A ?A-id ?B ?B-id)
(test (eq derived ?concept-name))
```

The required partial match interrogation function must therefore operate at a higher level of abstraction than the built-in CLIPS `matches`. We implemented this function by cannibalizing the CLIPS matches code, modifying it to return its results as functional return values rather than as printed output, and building a layer on top of this code to interpret the results in terms of Hendrix pattern conditions.

Unfortunately, we did not have time to create a menu item for invoking this capability, so in Hendrix 2.0 the user does not have access to it. Adding this interactive handle would be a trivial task for anyone with working knowledge of the XVT GUI-builder.

3.3 Refining the Transformation Engine

The Lessons Learned document made two recommendations concerning the execution of transformations. Although the transformation engine is not, strictly speaking, part of the learning capability itself, it is an essential part of Hendrix, without which the learning capability would be meaningless. As the design documentation in the Hendrix Implementor's Guide makes clear, it is a fairly complex piece of software.

The recommendations were to allow the user to focus composed transformations on a subset of elements in a specification, and to enable the firing of concept rules at all times during the execution of a transformation (rather than in batch mode between composed transformations, as was the case in Hendrix 1.0). The latter capability requires truth maintenance: Hendrix must now recognize when changes to a specification invalidate a concept that previously held.

3.3.1 Focus of Attention

When the user defines a complex transformation as a composition of previously defined transformations, he is now prompted to specify any desired constraints on the elements to which each transformation should be applied. These constraints are then stored along with the name of the component transformation itself, in the definition of the complex transformation.

The main complication in implementing this capability is the following: the user might focus a transformation on elements that did not exist in the original specification (from which the complex transformation proceeds) but that were created by the complex transformation itself, during an earlier step. This requires that every element in every phase of a specification's partial transformation be characterized in terms of the role it plays within the overall transformation.

To address this problem, the dependency information (discussed at the end of Section 2.3.2) is used to generate a functional description of each new element. This functional

description serves as the element's "role" in the transformation. For example, in the paraphrased transformation presented in Section 2.4, we see the notation

```
Casais#New[Child,Parent]
```

which is interpreted to mean:

> The element `New` created by the transformation `Casais` as a function of the elements `Child` and `Parent`

Such functional characterizations can be nested to any depth, depending on depth of composition of the component transformations.

The focus-of-attention capability allowed us to eliminate many artifacts needed in the Phase Two demonstration to teach Hendrix the Casais transformation. We no longer need to create intermediate "flags" in the evolving specification to keep track of where the overall recursive transformation "is," as it traverses the inheritance graph. These flags were, in the Phase Two demonstration, created and then erased by separate transformation steps which the user had to define. They effectively required the user to think in terms of procedural programming rather than in terms of the desired transformation itself. With the focus-of-attention capability, the user now defines the Casais transformation in the following form:

> Perform the following operations on the initial pattern: ...
> Then recurse up the inheritance graph
> When you return, continue with the following operations on the initial pattern ...

As a result, the process of teaching Hendrix the transformation is greatly simplified, and the system itself is much more usable.

3.3.2 Truth Maintenance

CLIPS 5.1 has a built-in truth maintenance capability, in which an asserted fact can be declared to be dependent on a condition; if the condition goes away, the fact will be retracted too. Unfortunately, the implementation of truth maintenance in CLIPS 5.1 does not behave correctly when the condition is a negation. Accordingly, we implemented explicit truth maintenance for Hendrix 2.0, contained in the file `hendrix.truth`.

The implementation involves the traversal of logical dependencies whenever a fact is asserted or retracted, and it is therefore rather slow (although with dribble turned off and a binary image of the rule base loaded, we expect the overhead will be not too bad). Since the built-in truth maintenance in CLIPS 6.0 does behave properly on negations, we have eliminated `hendrix.truth` from Hendrix 2.1 and have modified the pattern rules to employ the built-in truth maintenance. Besides being more efficient, this greatly simplifies the pattern rules themselves, which were saddled with a lot of additional baggage to establish the necessary logical dependencies between facts.

4 Directions for Future Work

In this section we briefly identify the areas in which we think continued work on Hendrix would be valuable.

4.1 Usability Improvements

In reconstructing the Hendrix Demonstration with the new system, we found that the tool is consdierable easier to use effectively than it was before the changes described in Section 3. Nevertheless, we still encountered some difficulties, which we summarize here.

Optional facts in a pattern. This enhancement, which is similar to irrelevant feature elimination in explanation-based learning, would be the single most effective improvement. It would allow us to collapse the diagrams in the Hendrix Demonstration to a small set, probably by another factor of two or more. The user should be able to designate selected items in a pattern exemplar (nodes, arcs, attributes, or tokens within a label) as "optional," meaning that they may or may not be present. When teaching Hendrix how to transform such a pattern, operations on the optional elements would be generalized to be conditional on the presence of the item.

With such an enhancement, we would no longer have to separate the initial steps in the Casais transformation into the five distinct, previously defined transformations:

```
copy-inherited-method
copy-virtual-method
copy-overridden-method
copy-disembodied-method
copy-grandparent
```

Each of these steps currently requires teaching Hendrix both a pattern and a transformation on that pattern. From the user's perspective, a more natural approach would be to include an exemplar of each type of method in the inital Casais pattern, and an exemplar grandparent, and indicate that (a) they may or may not be present, and (b) if they are present, they should be copied.

Optional facts in patterns and transformations could be implemented by having Hendrix generate helper rules along with the pattern and transformation rules. The helper rules would establish whether each optional element is present, and record this observation in a variable that can be tested by the actions of the transformation rule:

```
(defrule helper-1
     ($?fact)
=>
     (bind $?v-fact (mv-append TRUE ?fact))
     (assert (v-fact))
)
(defrule helper-2
     (not ($?fact))
=>
```

```
(bind $?v-fact (mv-append FALSE ?fact))
(assert (v-fact))
```

Generalizing over the number of tokens. Another source of irritation in teaching Hendrix the Casais transformation was the need to distinguish between deferred and implemented methods when the distinction was not relevant to the transformation. In our approach to representing the transformation, methods are specified as attributes of the class to which they belong. A deferred (or virtual) method is indicated by a single-token attribute, representing the name of the method. An implemented method is indicated by two tokens, the second of which is a label for the body (or implementation) of the method.

The problem is that Hendrix generates pattern rules to match the specific number of tokens found in the pattern exemplar.[6] The user has no direct way of telling Hendrix, "Copy this method to this other class, regardless of whether or not it has a body." Of course, certain operations in a transformation may rely on the specific number of tokens in an attribute or label; but copy operations do not, and Hendrix should be able to recognize such operations and generalize the number of tokens involved.

A striking example of this problem arose when we discovered, midway though reconstructing the demonstration, that we had forgotten about a special case that arises as an intermediate step in the Casais transformation. The pattern involved is that of a subclass overriding the method implementation defined by its parent. In this special case, the overriding consists of the subclass declaring the method to be deferred (i.e., no body). This is something that a human would not do, but it arises automatically as an intermediate state in the Casais transformation, and must be handled by successive steps in the transformation. What was particularly disturbing about this is that we made the same mistake initially in the Phase Two demonstration. The fact that we forgot about it indicates the unnaturalness of requiring the user to distinguish and enumerate such cases.

With number generalization, the user could define the pattern `method-overridden` in terms of a parent class that defines the method's body, and a child class that does not inherit this body. Whether the child class provides another body, or simply declares the method as deferred, would be irrelevant, and the distinction would not even have to occur to the user.

Predicting the required pegs. We found, in using Hendrix, that it is still difficult to predict when items in a pattern exemplar should be allowed to take on the same values. It may be more helpful for Hendrix to default to this condition, and require the user to specify when values must be distinct. On the other hand, this may lead to patterns in which the values are necessarily distinct but because Hendrix has not been informed of this, CLIPS has to attempt exponentially many unsuccessful matches. It was the performance degradation resulting from this phenomenon in the earliest version of Hendrix that led us to the default assumption of distinctness. This issue needs further research.

Defining continuation transformations on the fly. Currently, if the user wants to teach Hendrix a complex transformation — one in which a series of edit operations is followed by the composition of several other transformations — she must ensure that the composed transformations have been previously taught to Hendrix. This feels unnatural because it destroys the sense of operating on a specific exemplar, requiring instead that the

[6] There is one exception to this: a node or arc with a single-token label in a pattern exemplar will, in the generated pattern rule, match nodes or arcs with multi-token labels.

user think out the complex transformation in abstract, hierarchical terms. On the other hand, it may be that this usability problem will disappear with the "optional element" enhancement discussed above, since most of the continuation transformations in the demonstration were necessitated by the absence of optional elements.

Relationships between labels. We mentioned in Section 3.1.4 that the user can now specify relationships between labels by having the labels share one or more tokens. Any such relationships are recognized and preserved by the transformations Hendrix learns.[7] A more refined capability would recognize the identify not just of shared tokens, but of substrings within individual tokens. In the example cited in Section 3.1.4, this would enable us to retain the underscore in the label `<object>_ASSIGNER`, where currently we have to replace it by a space.

Such an enhancement would require distinguishing between coincidental identity of substrings, and intended identity — another form of irrelevant feature elimination. Presumably some form of user input would be needed to make this distinction, and this in turn raises issues of intrusiveness. If, however, we try to extend Hendrix to support program transformations in the real world of software development, such an enhancement will probably be necessary: a great deal of implicit design information is typically encoded in the structure of variable, function, and other names in a program.

4.2 Learning Code Generation Rules

Code generation is a form of design trransformation, and it seems natural to apply Hendrix techniques to support the rapid creation of customized code generators. We performed some initial experiments teaching Hendrix code generation rules for state machines, hoping eventually to have Hendrix learn how to generate its own internal state machine code. The experiments have convinced us that this is a feasible and valuable use of the tool, although some of the usability enhancements discussed above would have to be made for it to be a practical tool.

4.3 Hendrix as a Maintenance Tool: Learning Program Transformations

Another very inviting application of Hendrix is as a maintenance assistant. Source code maintenance frequently involves relatively rote but time-consuming modifications across numerous software modules. For example, adding a menu item to Hendrix involves changing the XVT resource file, and then creating various interface functions that every such item has but whose source code content is dependent on the particular menu item. If a maintainer could perform such a change once, indicate to Hendrix what aspects of the change are parameterizable, and have Hendrix generate a transform, he could be freed of one time consuming task after another. The Hendrix technology is not yet ready for this application, since the issues of irrelevant feature elimination need further study. But we believe it is a promising avenue that could have real impact on the industry.

[7] With a limitation: relationships between old elements, or between old and new elements, are recognized; those between new elements are not.

4.4 Integration with the ADM

Finally, it would be desirable to integrate Hendrix with the ADM to show its viability as a part of the overall KBSA. The ADM has been designed to facilitate the integration of tools, in a loosely coupled mode. In the case of Hendrix, we would probably want a closer form of integration which enabled the ADM to learn its own rules. This would involve constructing some kind of mediator between the Hendrix CLIPS representation and the ADM's representations, and replacing the Hendrix user interface with that of the ADM. Of these two tasks, the mediation is likely to present the most challenges.

References

Basili, V. "The Experience Factory and its Relationship to Other Improvement Pradigms." Keynote Address, *European Conference on Software Engineering*, 1993.

Casais, E. *Managing Evolution in Object Oriented Environments: An Algorithmic Approach.* Ph.D. Dissertation, Faculty of Economic and Social Sciences, University of Geneva, 1991.

Coleman, D. et al. *Object-Oriented Development: The Fusion Method.* Prentice Hall, 1994.

DeJong G., Mooney, R. Explanation-based learning: an alternative view. *Machine Learning* **1**, pages 145-176. 1986.

Lieberherr, K., Holland, I., Riel, A. Object-oriented programming: an objective sense of style. *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '88).*, Special Issue of SIGPLAN Notices. ACM Press, November 1988, pages 323 - 334.

Lieberherr, K., Bergstein, P., Silva-Lepe, I. From objects to classes: algorithms for optimal object-oriented design. *Software Engineering Journal*, BCS/IEE, July 1991, pages 205 - 228.

Partsch, H. *Specification and Transformation of Programs.* Spinger-Verlag, 1990.

Petroski, H. *To Engineer is Human.* Random House, 1992.

☆U.S. GOVERNMENT PRINTING OFFICE: 1996-710-126-47035

MISSION

OF

ROME LABORATORY

Mission. The mission of Rome Laboratory is to advance the science and technologies of command, control, communications and intelligence and to transition them into systems to meet customer needs. To achieve this, Rome Lab:

a. Conducts vigorous research, development and test programs in all applicable technologies;

b. Transitions technology to current and future systems to improve operational capability, readiness, and supportability;

c. Provides a full range of technical support to Air Force Materiel Command product centers and other Air Force organizations;

d. Promotes transfer of technology to the private sector;

e. Maintains leading edge technological expertise in the areas of surveillance, communications, command and control, intelligence, reliability science, electro-magnetic technology, photonics, signal processing, and computational science.

The thrust areas of technical competence include: Surveillance, Communications, Command and Control, Intelligence, Signal Processing, Computer Science and Technology, Electromagnetic Technology, Photonics and Reliability Sciences.